MDL SOFTWARE TESTING GUIDELINES

1.0 INTRODUCTION

This Meteorological Development Laboratory's (MDL) Test Guidelines (TG) describes objectives, strategy, validation techniques, and documentation standards used for testing the MDL software applications. The purpose of this document is to provide a standard approach for testing all MDL applications. The goal is to provide an effective, repeatable, software test process which is independent of the software language, design methodology, and development environment. The scope of the testing described in this document is to identify and remove all defects and also to validate that all software applications meet all requirements allocated to them while being implemented in a correct and efficient manner. This document is organized as follows:

Section 2.0 - defines the objectives of testing and defines test verification methods

Section 3.0 defines the test planning activity

Section 4.0 defines testing documentation such as test procedures, and test logs

2.0 OBJECTIVES AND TEST VALIDATION TECHNIQUES

The overall objective of testing is to verify compliance of the software with it's requirements. Software test activities are driven by the need to verify that the software satisfies all of it's requirements. Each software requirement is verified by one or more of the following methods:

Inspection - A technique in which satisfaction of the requirement is verified by inspection of hardware, source code and or other physical means.

Test - A technique in which satisfaction of the requirement is verified by exercising the software and recording and analyzing the results of the test.

Demonstration - A technique in which satisfaction of the requirement is verified by observing the performance of the software.

Analysis - A technique in which satisfaction of the requirement is verified by inference based on examination and analysis of the internal structure of the software and it's components. This may be required when a requirement cannot be directly tested and observed because of a unique configuration not available on the test platforms.

Simulation - A technique in which satisfaction of the requirement is verified by using a representation of a missing component (e.g., a specifically developed component of code to represent a hardware component.

3.0 TEST STRATEGY

The primary goal of MDL's testing activities is to identify and remove defects from individual application software, before allowing them to proceed to later levels of system integration, and to ensure that all MDL software requirements traceable to the applications are satisfied. Our test strategy has two basic points of view. The first, termed **functional**, specifies testing from the perspective of a functional specification without concern for the specific implementation details. This type of testing takes on the user's point of view at the application level. At the unit and module levels, functional testing is only concerned with the functional interface and module behavior from the perspective of a calling function outside of the entity to be tested.

The second testing point of view, termed **structural**, focuses on studying the implementation details embodied in the source code to determine program attributes and correctness. Structural testing can consider such programming and implementation details as logic paths, language characteristics, and compliance with programming standards. There are times when the structure of the code helps determine the functional test that needs to be performed, and the boundaries between functional and structural testing can become fuzzy. Testing of each application is designed to include adequate coverage of both the functional and structural aspects.

Testing MDL software applications is done in two steps--informal, then formal. The dividing line between these two steps is simply the point at which the application code is handed over to configuration management (CM) for incorporation into the baseline. Testing of code performed prior to submitting to CM is considered informal because testing activities and code development can be performed without any formal actions outside of the development team. Once code is under CM control, testing is considered formal because all activities involving the base-lined code, such as testing and changing the code, must be formally documented (defect reports, change documents) and controlled by CM.

The testing process follows a bottom-up approach, although some aspects of top-down testing are also incorporated by using high level drivers which eventually evolve into modules.

Testing begins at the lowest unit level and proceeds upward as units are integrated to form modules. At the unit/module level, testing is required for all modules and is performed in the developers own workspace. Once the unit/module level development and testing is complete, software integrated testing begins at the application level to test the integrated functionality of the modules. Software integration testing takes place in the development testing environment. Once the software integration testing is completed the application is loaded onto the operational testing environment to test compatibly with other current applications.

Appendix 1 defines the various techniques that should be considered for each of the testing levels being applied to verify the MDL application software.

3.1 TEST PROCESS

The testing process follows a bottom-up approach, although some aspects of top-down testing are also incorporated by using high level drivers which eventually evolve into modules. Testing begins at the lowest unit level and proceeds upward as units are integrated to form modules. At the unit/module level, testing is required for all modules and is performed in the developers own workspace. Once the Unit/Module Testing is complete, Software Integrated Testing begins at the application level to test the integrated functionality of the modules. Software Integration Testing (SwIT) takes place in the

development testing environment. Once the SwIT is completed and an Integration Readiness Review (IRR) is successfully completed, the application is loaded onto an operational testing environment for System Integration Testing (SIT) to test compatibly with other current applications. SIT testing is formally ended upon completion of the Operational Readiness Review (ORR).

3.1 UNIT/MODULE TESTING

The purpose of unit/module testing is to provide early identification of internal errors in such aspects of the software as functionality, program structure, and performance. Units and modules are tested for:

- limit and range validation (boundary analysis),
- error handling validation,
- logic/path validation,
- local data structure validation,
- performance testing,
- application programming interface validation, and
- user interface style guide compliance.

This level of testing is performed soon after a unit is compiled successfully or a module is built successfully and the appropriate code analysis utilities (e.g., Purify and Lint) have been applied to the unit. A peer review or complete code walk through is conducted by the developer with an appropriate technical individual(s) prior to testing to ensure the code fulfills technical and functional requirements and complies with MDL software standards.

Unit/Module testing is performed by the developer using a software debugger or test drivers which exercise the units and modules. Unit/module testing is performed within the developer's own workspace in the development environment. Developers are responsible for identifying and documenting unit test procedures to fully verify the software. Each test procedure, its test data (if used), and its test results will be documented in the Software Documentation Files (SDF) using the test procedure forms adopted by the project.

As a final step, the developer may perform validation testing on appropriate modules. Validation testing involves comparing the output of the module with that of an existing piece of software known to produce correct output. This testing will be performed only when comparable data are available for the module.

Unit testing is considered complete after all of the test procedures have been successfully executed and the code walkthrough has been conducted. Appendix 2 provides some Unit Test Guidelines and Appendix 3 contains a Unit Test Checklist.

3.2 SOFTWARE INTEGRATION TESTING

The purpose of software integration testing is to detect functional errors, and demonstrate interface compatibility by testing the integrated application. in an environment which simulates operational capabilities. This will involve testing the integration of all software modules which comprise the application. In addition, regression testing is conducted upon implementation of any system/software changes. Regression testing is a subset a previously executed tests, the level of which is proportional to the scope and impact of the changes.

The software integration testing takes place after the application has been unit tested and a code walkthrough has been conducted. To begin this phase of the testing process, the developer(s) must identify functional test procedures to validate the software requirements. These may be derived from the same test procedures used for unit testing, but must address the integrated aspects of the application. For each test procedure, the developer must write the test descriptions, obtain test data, and schedule a review of the test procedure. For large pieces of code involving new functionality, a Test Readiness Review should be held to verify the completeness of the test procedures before approving the software for software integrated testing. For small enhancements or bug fixes it is sufficient for the Task Lead or their appointee to review the test procedures before approving the associated change document.

The SwIT is considered complete when all of the associated test procedures are passed and all pending or failed test procedures are reassigned to a developer for repair. Once testing is completed, an Integration Readiness Review (IRR) is held to verify the completeness of the test procedures before approving the software for System Integration Testing (SIT).

3.3 SYSTEM INTEGRATION TESTING

The purpose of system integration testing is to independently validate the software requirements in an environment which simulates WFO operational capabilities. System Testing consists of a collection of subtypes, including load/stress, volume, performance, reliability/availability, fail-over, configuration, compatibility, security, instability, and human factors. This level of testing is performed after the software deliverables have been baselined under CM control, successfully passed all software integrated testing, and installed in the operational environment. To install the application, the Release Coordinator will use the same installation media used for the final software integrated test.

System Integration Testing is considered complete after all of the test procedures have been successfully executed and all pending SPRs have been reassigned to another release. An Operational Readiness Review is conducted before the software becomes operational.

4.0 TEST DOCUMENTATION

MDL Test Documentation includes a test plan, test procedures, and test log.

4.1 TEST PLAN

The testing process includes the careful and detailed analysis of system requirements and design, test bed constraints, and development tools that can be used for test instrumentation. The key test planning activities are:

- Defining the overall testing strategy and required tools to support it
- Designing tests, test case and test procedures to implement the strategy
- Specifying the overall system acceptance test pass/fail criteria
- Developing a schedule and staffing to accomplish the testing.

This information should be developed and documented in the project's software Test Plan. A test plan should be prepared for testing each new release of software. The Test Plan should identify:

- Focus
  - Integration Points
  - New Data Interfaces
  - New Functionality
- Test Procedures for the following categories
  - Functionality
  - Integration
  - Regression
  - Fail over
  - Stress
  - Performance
  - Reliability
  - Ad-hoc
- Test environment
  - Hardware and Commercial-Off-The-Shelf (COTS) software
- Test Tools (if any)
- Schedule and Staffing

4.2 TEST PROCEDURES

Regardless of the level or stage of testing, the basic structure of the testing will be essentially the same. For a given unit, module, or application, one or more test procedures will be identified to evaluate the functional or structural condition of the code. Test procedures will be designed based on specific functional requirements or components of code structure.

Each test procedure must include:

- a test description
- test type
- required support software (e.g., drivers, stubs, tools), inputs
- expected outputs
- compilation of the test results
- identify the software requirements validated by the test

Once the individual test procedures have been created, they will be kept under a Unit/Module folder and a Master List of all the test procedures will be kept in the SDF.

To determine the success or failure of a test, the tester will conduct any necessary data reduction or analysis and compare the actual results with the expected results. For each discrepancy, the tester will try to determine if the problem is in the test, the software being tested, or the hardware. How the problem is handled depends on whether or not the testing is being done formally or informally. If the testing is informal, the problem will be fixed immediately and testing will be continued. If the testing is formal, then the problem will be formal documented and submitted to CM before changes can be made to the software. Appendix 4 contains a template for a Test Procedure.

5.3 TEST LOG

A test log should be prepared for each project/task for each test cycle and release. A test log contains a master list of test procedures, responsible tester, when it was tested, pass/fail, comments and a summary of the test cycle results. Appendix 5 contains a template for a test log. A test log should be completed for each project/task.

APPENDIX 1
TESTING TECHNIQUES

This section defines the various techniques that should be considered for each of the testing levels being applied to verify the MDL applications software. A summary of the affiliation between test levels and the test techniques is provided in Table A1-1 and Table A1-2. Not all test techniques are relevant to all types of software development projects. Software developers must be familiar with the various techniques and apply the appropriate technique and degree of testing as the project requirements dictate. It will be the responsibility of the person or team defining the test cases to choose the appropriate testing techniques employed at each level of testing.

Path Testing

Path testing is a highly structural test technique well-suited to unit/module level testing. Path testing consists of exercising selected executable paths in an effort to identify control structure errors. However, it is virtually impossible to test all paths through a software system or even all paths through a relatively simple unit. The solution is to carefully choose a small set of paths that provide complete coverage.

Criteria for path selection, at a minimum, requires that enough paths be chosen in order to ensure that every decision/condition point is taken in each possible direction at least once and that every line of code is executed at least once (complete coverage). Ensuring that every decision/condition point is taken in each possible direction at least once is equivalent to exercising all sub-elements from which all paths are created, without exercising all paths. Anything less than complete coverage results in untested code being integrated into the system.

Deriving the path-forcing input values is called sensitizing the path. Sensitizing can be accomplished in either a forward or backward direction, by starting at the beginning or the end of the path, respectively. In either case, the control structures are navigated by choosing the broadest range of values allowed without violating the previous choices. In this way, the entire path can be navigated using the set of sensitized values.

Boundary Condition Testing

Boundary condition testing can be applied to both structural and functional testing. It consists of identifying and using input values that exercise the maximum, minimum, and just-beyond maximum and minimum software tolerances. Both input and output tolerances should be exercised; for example, input values that yield the minimum and maximum value for the associated output parameter should be chosen, as well as input values that are themselves minimums and maximums.

Pure path testing alone does not detect certain types of errors, such as missing paths or incorrect logic. Therefore, to maximize error detection, path testing must be applied in combination with other techniques, such as boundary condition testing. Additionally, an understanding of what the path is intended to accomplish, including detailed expected results, is necessary to uncover invalid logic within valid paths.

Input Validation and Syntax Testing

Input validation and syntax testing is a common functional testing technique used to test the point where a human-computer interface interacts with the code. In order of importance, the priorities for generating input validation and syntax testing cases are as follows:

- test for direct effects of invalid input,
- test for indirect effects of invalid input, and
- test acceptance of valid input.

Transaction Flow Testing

Transaction flow testing techniques are similar to path testing. However, a transaction is a functional capability from the user's perspective. Therefore, transaction flow testing is performed on a functional level and is independent of implementation. Transaction flow testing will comprise the majority of test cases within any post integration testing activities.

Transaction flow path selection consists of analyzing the software requirements to determine a set of fundamental transactions that the software is required to perform. The design of these test cases should be able to detect invalid paths through the application as well as test valid transaction flows.

Equivalence Partitioning

Exhaustive input testing typically is not possible, particularly for functional oriented testing. Instead, functional-type testing is performed through a small subset of all possible inputs. Therefore, the selection of the appropriate subset becomes a critical element of test case design.

In identifying equivalence classes, both valid and invalid equivalence classes should be specified, where invalid classes represent erroneous input values and invalid states or results.

Equivalence partitioning, as a testing technique, should be combined with other techniques to both improve the quality of tests implemented and limit their number.

Table A1-1. Testing Techniques Applied to Unit/Module Level Testing

Test Level	**Technique**	**Purpose**	**Characteristics**
Unit/Module Level Testing	Path Testing	Execution of every logic branch and line of code finds logic errors in control structures, dead code, errors at loop boundaries, and errors in loop initializations.	Perform McCabe's cyclomatic complexity analysis to help determine number and focus of unit and module test cases. Perform path sensitization to determine test case parameters. Choose parameter which complement other techniques, such as boundary conditions and invalid syntax inputs.
	Boundary condition testing	Interface testing finds errors in input and output parameter tolerances and verifies the program limits are correctly stated and implemented.	Test tolerances such as parameter minimums, maximums, and "just beyond" minimums and maximums. Choose input parameter that test both input and output tolerances.
	Input validation and syntax testing	Verifies that the error handling facilities of the program operate as stated and that these facilities are sufficient for the errors that occur. Valid and invalid inputs uncover errors in the user interface module under test.	Force every error message and verify the accuracy and clarity of each. Choose valid and invalid input parameters. Invalid parameters include wrong type, scope, length and special keyboard characters, ESC, CNTL, etc.

Table A1-2. Testing Techniques Applied to Software and System Integration - Testing.

Test Level	**Technique**	**Purpose**	**Characteristics**
Software and System Integration Testing	Boundary conditions testing	Interface testing finds errors in input ans output parameter tolerances and verifies that the program limits are correctly stated and implemented.	Test tolerances such as parameter minimums, maximums, and "just beyond" minimums and maximums. Choose input parameters that test both input and output tolerances.
	Input validation and syntax testing	Verifies that the error handling facilities of the program operate as stated and that these facilities are sufficient for the errors that occur. Valid and invalid inputs uncover errors in the user interface module under test and side effects that show up in other related modules being integrated.	Force every error message and verify the accuracy and clarity of each. Choose valid and invalid input parameters. Invalid parameters include wrong type, scope, length and special keyboard characters, ESC, CNTL, etc.
	Equivalence partitioning	Reduce necessary number of test cases for adequate coverage. Uncover functional errors in the execution of integrated modules.	Improve probability of uncovering errors versus random cases by partitioning test cases by class. Choose representative set of cases that cover all classes.
	Transaction flow testing.	Uncover functional and performance errors in the execution of integrated modules. Verifies proper functional capability against all software requirements.	Similar steps as in path testing but a functional /performance level. Test for valid and invalid paths. Perform transaction flow path selection using software requirements from the perspective of application users.

APPENDIX 2
UNIT TESTING GUIDELINES

The role of unit testing in the Post Build 5 process is a focus on the implementation of the design and is an important complement to the System Integration Test and functional Evaluation Test processes. These guidelines and checklist are intended to provide guidance on the areas that will be presumed to be unit tested upon entrance to the test phase following development. It is not intended to replace the peer review process or the code walkthrough, but rather serve as an additional tool to assist in improving the quality of AWIPS software. Specific guidelines:

1. Unit tests should be conducted on all new or modified modules (i.e., function, subroutine or class). The extent of the testing should be guided by the complexity of the module.

2. Unit tests should be planned and conducted to take into account design assumptions, a full range of operational possibilities and special cases, where appropriate.

3. A complete unit test is comprised of both positive and negative test cases. Positive test cases should include all values and conditions expected to be encountered. Negative test cases should include, to the extent possible, nvalid and/or missing values and conditions.

4. A complete unit test will verify all input and output parameters conform to design assumptions.

5. A complete unit test will execute each statement in the module, at least once.

6. At a minimum, test cases should be comprised of a subset of the test factors included on the attached checklist. Any additional test requirements imposed by individual development organizations should be included, as well.

7. Tests of boundary condition values should be tested for minimum, minimum -1, minimum +1, maximum, maximum -1 and maximum +1 values.

APPENDIX 3
UNIT TEST CHECKLIST

Test Factor	**N/A**	**Values Checked**
Boundary Checks		
Date Parameters		
Occurrence Parameter		
Value Parameter		
Formula Parameter		
Common Valid Representations		
Leading Zeros and not		
Leading Blanks and not		
Record Type combinations		
Left or Right Justification		
Existence of relationship(s) Header/trailer)		
Order/sequence		
Computation/Total		
Transaction Types		
Transaction Combinations		
Other		
Invalid Input Selections		
Numeric Values - Sets and Ranges		
Invalid Blanks		
Invalid Signs		
Non-numeric		
Too large/Too small		
Invalid Middle for sets		
Invalid Characters		
Missing key fields		
Invalid key field		
Header record not initial record		
Trailer record not last record		
Invalid or missing header/trailer record		
Data Structures		
Missing required elements		
Extraneous unknown elements		

Test Factor	**N/A**	**Values Checked**
Extraneous duplicate elements		
Invalid Sequence		
Inconsistent value		
Insufficient size		
Excessive size		
Invalid combinations		
Termination		
Normal Termination		
Abnormal Termination		
Outputs Normal		
Reports		
Files		
Transmissions		
Queries		
Outputs Nil		
Reports		
Files		
Transmissions		
Queries		
Algorithms and Computations		
Database Transaction		
Event driven		
Time driven		
All Error Messages (not covered with negative test)		
GUI Components		

APPENDIX 4
TEST PROCEDURE TEMPLATE

PROJECT NAME:

Module/Unit Name:

Test Procedure Title: **Test Procedure ID:**

Test Environment:

Location	Machine Name	OS Version	AWIPS Version	Application Version
NOAA/NWS HQ Silver Spring, MD				

Test Result:

Test Date	Tester Name	Start and End Time	Status (Pass/Fail)	Comments

Test Procedure Update History:

Update Requested by	Date Changed	Comments

Requirements Addressed:

Prerequisite Conditions:

General Test Steps Needed for this Test Procedure:

Test Description:

ID	Step	Expected Result	Pass/Fail
1			
2			
3			
4			
5			
6			
7			
8			

APPENDIX 5
TEST LOG TEMPLATE

PROJECT NAME TEST LOG:

Master List of Test Procedure / Testing Cycle Results				
Total Number of Test Procedure :	**8**	**Percent Tested:**	**50.00**	
Total Pass :	**3**	**Percent Pass:**	**75.00**	
Total Fail :	**1**	**Percent Fail:**	**25.00**	
Total Tested:	**4**			
Test Procedure ID and Title	***Tester Name***	***Test By Date***	***Pass/Fail***	***Comments***
Module/Unit Name				
Test Procedure 1	Tester 1		Pass	
Test Procedure 2	Tester 2		Fail	
Test Procedure 3	Tester 3		Pass	
Test Procedure 4	Tester 4		Pass	
Test Procedure 5				
Module/Unit Name				
Test Procedure 1	Tester 1			
Test Procedure 2	Tester 1			
Test Procedure 3	Tester 1			
Test Procedure 4	Tester 1			